Lecture 11

Complex Data Types: Structures, Enumerators, and Unions

Fundamentals of Computer and Programming

Instructor: Morteza Zakeri, Ph.D. (m-zakeri@live.com)

Spring 2024

Modified Slides from Dr. *Hossein Zeinali* and *Dr. Bahador Bakhshi*

Computer Engineering Department, Amirkabir University of Technology

What We Will Learn

- Introduction
- **`struct`** definition
- Using **`struct`**
  - **`struct`** and Array
  - **`struct`** and Pointers
  - **`struct`** and Functions
- Linked-List
- **`enum and unions`**

What We Will Learn

- Introduction
- `struct` definition
- Using `struct`
  - `struct` and Array
  - `struct` and Pointers
  - `struct` and Functions
- Linked-List
- `enum` and `unions`

Introduction

- Our variables until now
  - Single variable

```
int i, char c, float f
```

  - Set of same type elements: Array

```
int a[10], char c[20]
```

- If data are not *same type*, but related? Example: Information about students
  - Student Name
  - Student Family Name
  - Student Number
  - Student Grade

Introduction

- How to save the student information?
- **1.** Use separated variables
  - `char st_name[20];`
  - `char st_fam_name[20];`
  - `int id;`
  - `int grade;`
- **2.** Put them altogether, they are related
  - Use `struct`
  - This concept is extended in OOP as the "object"

What We Will Learn

- Introduction
- **`struct`** definition
- Using **`struct`**
  - **`struct`** and Array
  - **`struct`** and Pointers
  - **`struct`** and Functions
- Linked-List
- **`enum and unions`**

`struct`: version 1

- Set of related variables
  - Each variable in **`struct`** has its own type

- **`struct`** in C (version 1)

```
struct {
    <variable declaration>
} <identifier list>;
```

`struct` (version 1): Example

```
struct{
   char st_name[20];
   char st_fam_name[20];
   int id;
   int grade;
} st1;
```

- We declare a variable `st1`
- Type of `st1` is struct
- `id` is a member of the struct
- `grade` is a member of the struct

`struct` (version 1): Example

```
struct{
   char st_name[20];
   char st_fam_name[20];
   int id;
   int grade;
} st1, st2, st3;
```

- We declare three variables: `st1, st2, st3`
- Type of `st1, st2, st3` is the struct
- In this model, we cannot reuse the struct definition in other location (*e.g.*, input of function)

`struct`: version 2

- **`struct`** in C (version 2)

```
struct <tag> {
    <variable declaration>
};

struct <tag> <identifiers>;
```

`struct` (version 2): Example

```
struct std_info{
   char st_name[20];
   char st_fam_name[20];
   int id;
   int grade;
};

struct std_info st1, st2, st3;
```

- We define a struct with tag `std_info`
  - We do not allocate memory, it is just definition
- We declare variables `st1, st2, st3` from `std_info`

`typedef`

- We can assign a new name for each type
  - Assign name “integer” to “int”
  - Assign name “int_array” to “int[100]”
  - Assign name “int_pointer” to “int *”
- New names are assigned by **`typedef`**
- After we assigned the new name, we can use it in identifier declaration

`typedef`: Examples

```
/* Assign new name integer to type int */
typedef int integer;

/* Use the new name */
integer i, j, k;

/* Assign new name alephba to type char */
typedef char alephba;

/* Use the new name */
alephba c1, c2;
```

typedef: Examples

```
/* Assign new name intptr to type int * */
typedef int * intptr;

/* Use the new name */
intptr pi, pj, pk;

typedef int int_arr1[10], int_arr2[20];

int_arr1 array1;
int_arr2 array2;
```

`struct`: version 3.1

- Using the typedef

```
struct <tag>{
   <variables>
};

typedef struct <tag> <new_name>;

<new_name> <variables>;
```

`struct` (version 3.1): Examples

```
struct std_info{
  char st_name[20];
  char st_fam_name[20];
  int id;
  int grade;
};

typedef struct std_info information;

information st1, st2;
```

`struct`: version 3.2

- struct in C (version 3.2)
- Using the typedef

```
typedef struct{
   <variables>
} <new_name>;

<new_name> <variables>;
```


`struct` (version 3.2): Examples

```
typedef struct {
  char st_name[20];
  char st_fam_name[20];
  int id;
  int grade;
} information;

information st1, st2;
```

Structures as New Data Type

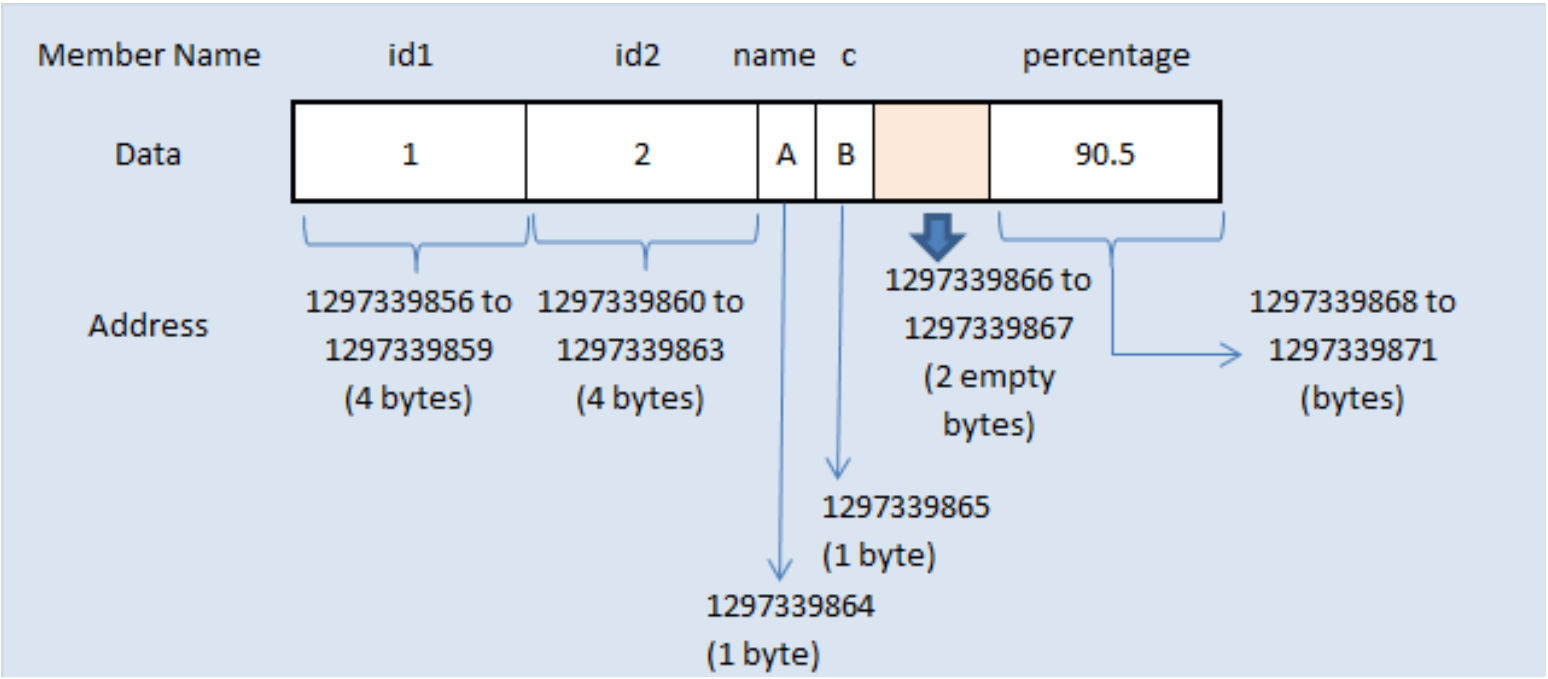
- When we define a new **`struct`**, in fact we are **defining a new data type**
  - Then we use the new data type and define variables
- So, we need to learn how to work it
  - Access to members
  - Operators for **`struct`**
  - Array of **`struct`**
  - **`struct`** in functions
  - Pointer to **`struct`**

Size of struct

- The size of struct is NOT the sum of size of members!
  - `struct test_size{char c, int i}`
  - `sizeof(struct test_size) // = 8 (!!!)`
- This is because of "Structure Padding"
  - Computer hard ware cannot (should not) read any arbitrary address
  - The address should be aligned in word
    - 4 bytes in 32-bit machine
  - The padding is to align the address
  - More details and examples: *https://fresh2refresh.com/c-programming/c-structure-padding/*

Example for structure padding in C language

```
struct structure1 {
        int id1;
        int id2;
        char name;
        char c;
        float percentage;
};
```



What We Will Learn

- Introduction
- `struct` definition
- Using **`struct`**
  - **`struct`** and Array
  - **`struct`** and Pointers
  - **`struct`** and Functions
- Linked-List
- `enum`

Using `struct`

- We should declare variables from `struct` type
  - Versions 1, 2, 3.1, 3.2

- How to access to the members of struct
  - <struct variable>.<element name>
  - **`st1.st_name`** is a array of char in struct st1
  - **`st2.grade`** is a int variable in struct st2

`struct` initialization

- Similar to array initialization

```
information st1 = {"Ali", "Karimi", 9222, 10};
```

  - “Ali” is assigned to st_name,
  - “Karimi” is assigned to st_fam_name,
  - 9222 is assigned to id,
  - 10 is assigned to grade,

- Order of values should be exactly the order of the members
- The number of values should be <= the number of members
- Initial values cannot be assigned in struct definition

Using `struct`

مثالی ساده برای نحوه استفاده از struct

```
#include <stdio.h>

typedef struct{
  char name[20];
  char fam_name[20];
  int id;
  int grade;
} information;

void main(void){
  information st2, st1 = {"Ali", "Hassani",
  90131, 20};

  printf("After init: \n");
```

Using `struct`

```
printf("Name = %s, \nFam. Name = %s, \nid = %d,
\ngrade = %d\n", st1.name, st1.fam_name,
st1.id, st1.grade);

scanf("%s", st2.name);

scanf("%s", st2.fam_name);

scanf("%d", &st2.id);

scanf("%d", &st2.grade);

printf("Your Input is: \n");

printf("Name = %s, \nFam. Name = %s, \nid = %d,
\ngrade = %d\n",

st2.name, st2.fam_name, st2.id, st2.grade);

}
```

Nested `struct`

```
struct date_type{
    int rooz, mah, sal;
};

typedef struct{
    char name[20];
    char fam_name[20];
    int id;
    int grade;
    struct date_type date;
} information;
```

Nested `struct`

```
information st1 = {"A","B",1,10,{2,3,1368}};


information st2;
st2.name = "C";
st2.fam_name = "D";
st2.id = 2;
st2.grade = 15;

st2.date.rooz = 10;
st2.date.mah = 5;
st2.date.sal = 1390;
```

`struct`: Copy and Assignment

```
struct date_type{
    int rooz, mah, sal;
};
struct date_type d1, d2 = {2, 1, 1360};

d1 = d2;
                /* d1.rooz = d2.rooz;
                   d1.mah = d2.mah;
                   d1.sal = d2.sal;
                */
```

`struct`: Copy and Assignment

```
struct test_type{
    char name[10];
    int id[10];
};

struct test_type d1, d2 = {"ABC", {1, 2,
3}};

d1 = d2;
                /* d1.name = "ABC";
                   d1.id   = {1, 2, 3};
                */
```

`struct`: Comparing

- We cannot compare **`struct`** variables
  - ==, <=, <, >, >= cannot be used for struct

```
information st1, st2;

if(st1 <= st2){    // Compile Error
   ...
}
```

- Why?
  - What does this mean? st1 <= st2

`struct`: Comparing

- We can compare members of structs

```
if((st1.id == st2.id) && (strcmp(st1.name,st2.name) == 0)
  &&
   (strcmp(st2.fam_name,st2.fam_name) == 0)){
   /* st1 == st2 */
}
```

- We can define <, <=, >, >= for struct

```
if((st1.id > st2.id) && (strcmp(st1.name,st2.name) == 0) &&
   (strcmp(st2.fam_name,st2.fam_name) == 0)){
   /* st1 > st2 */
}
```

`struct`: Arithmetic operations

- No arithmetic operation (+, -, /, …) is defined for structures
- We can define ours operations
- We have an example in the following slides

What We Will Learn

- Introduction
- `struct` definition
- Using `struct`
  - `struct` and Array
  - `struct` and Pointers
  - `struct` and Functions
- `enum` and `unions`

Array of `struct`: Definition

- struct is a type → We can define array of struct

```
struct std1{
   int id;
   int grad;
};
struct std1 std_arr[20];

typedef struct{
   int id;
   int grad;
} std2;
std2 std_arr[20];
```

Array of `struct`: Example

```
#include <stdio.h>
int main(void){
   struct std{
      int id;
      int grade;
   };
   const int num = 25;

   double sum, average;
   int i;
   struct std std_arr[num];

   for(i = 0; i < num; i++){
      printf("Enter ID and grade\n");
      scanf("%d", &(std_arr[i].id));
      scanf("%d", &(std_arr[i].grade));
   }
```

برنامه‌ای که شماره و نمره دانشجویان را بگیرد و لیست دانشجویانی که نمره آنها بیشتر از میانگین است را تولید کند.

Array of `struct`: Example

ادامه

```
   sum = 0;
   for(i = 0; i < num; i++)
      sum += std_arr[i].grade;

   average = sum / num;

   for(i = 0; i < num; i++)
      if(std_arr[i].grade >= average)
         printf("Student %d passed \n",
               std_arr[i].id);

   return 0;
}
```

Array of `struct`: Example 2

```
#include <stdio.h>

int main(void){
   struct std{
       char name[20];
       int id;
       int grade;
   };

   const int num = 25;
   struct std std_arr[num];
   int sid, i;

   for(i = 0; i < num; i++){
       printf("Enter Name, ID and grade\n");
       scanf("%s", std_arr[i].name);
       scanf("%d", &(std_arr[i].id));
       scanf("%d", &(std_arr[i].grade));
   }
```

برنامه‌ای که یک لیست از دانشجویان را بگیرد. سپس یک شماره دانشجویی بگیرد و اگر دانشجو در لیست است اطلاعات وی را نشان دهد.

Array of `struct`: Example 2

ادامه

```
  printf("Enter Search ID: ");
  scanf("%d", &sid);

  for(i = 0; i < num; i++)
      if(std_arr[i].id == sid){
            printf("Found:\n");
            printf("Name = %s\n", std_arr[i].name);
            printf("ID = %d\n", std_arr[i].id);
            printf("Grade = %s\n", std_arr[i].grade);
      }
  return 0;
}
```

What We Will Learn

- Introduction
- `struct` definition
- Using `struct`
  - `struct` and Array
  - **`struct`** and Pointers
  - `struct` and Functions
- **`enum` and `unions`**

Pointer to `struct`: Definition

- A variable of struct type is a variable
- It has address, we can have pointer to it

```
struct std{
   int id;
   int grade;
};
struct std st1;
struct std *ps;
ps = &st1;
```

Pointer to **`struct`**: Usage (version 1)

- We can use *pointer method
- **`*ps`** means the content of the address that **`ps`** refers to there → it is struct
- **`(*ps).id`** is the member of struct that `ps` refers to it
- **`(*ps).grade`** is the member of struct that `ps` refers to it
- **`*ps.id  // Compile Error`**

Pointer to `struct`: Usage (version 2)

- We can use "`->`" method

```
struct std{
      int id;
      int grade;
};

   struct std st1, *ps;
   ps = &st1
   int y = ps->id;       // (*ps).id
   int z = ps->grade;    // (*ps).grade
```

What We Will Learn

- Introduction
- `struct` definition
- Using `struct`
  - `struct` and Array
  - `struct` and Pointers
  - **`struct`** and Functions
- **`enum` and `unions`**

`struct` and Functions

- **`struct`** is a type → It can be used
  - In input parameter list of functions
    - Call by value
    - Call by reference
  - In return type of functions

```
void f(struct std s1);     // call by value input

void g(struct std *s2);    // call by reference

struct std h(void);        // return type
```

`struct` and Functions: Example

- struct as call by value input parameter

```
void print_st_info(information st){
  printf("Name = %s\n", st.name);
  printf("Fam = %s\n", st.fam_name);
  printf("id = %d\n", st.id);
  printf("grade = %d\n", st.grade);
}
//---- Calling the function ----
information st1;
print_st_info(st1);
```

`struct` and Functions: Example

- struct as call by reference input parameter

```
void read_st_info(information *pst){
   scanf("%s", pst->name);
   scanf("%s", pst->fam_name);
   scanf("%d", &(pst->id));
   scanf("%d", &(pst->grade));
}
//---- Calling the function ----
information st1;
read_st_info(&st1);
```

`struct` and Functions: Example

- struct as output of function

```
information create_st_info(void){
   information tmp;
   scanf("%s", tmp.name);
   scanf("%s", tmp.fam_name);
   scanf("%d", &tmp.id);
   scanf("%d", &tmp.grade);
   return tmp;
}
//---- Calling the function ----
information st1;
st1 = create_st_info();
```

Scope of `struct` definition

- A struct can be used only
  - In the defined scope
  - After definition
- → if **`struct`** is defined in a function
  - It can be used only in the function
  - No other function knows about it
- → If **`struct`** is defined as a global
  - It can be used in all function after the definition

Scope of struct variables

- The scope of struct variables are the same as other variables
- If struct variable is global
  - Initialized to zero and visible to the functions after its declaration
- If struct variable is automatic local
  - There is not any initial value, destroyed when the block finishes
- If struct variable is static
  - Kept in memory until program finishs

Example: Rational numbers

تابعی که دو عدد گویا را می‌گیرد و حاصل جمع و تفریق آنها را تولید می‌کند.

```
struct guia{
   int sorat, makhraj;
};

void f(struct guia a, struct guia b, struct guia *
   tafrigh, struct guia * jaam){
   int mokhraj_moshtarak = a.makhraj * b.makhraj;

   int sub = a.sorat * b.makhraj - b.sorat * a.makhraj;
   int sum = a.sorat * b.makhraj + b.sorat * a.makhraj;

   tafrigh->sorat = sub;
   tafrigh->makhraj = mokhraj_moshtarak;

   jaam->sorat = sum;
   jaam->makhraj = mokhraj_moshtarak;
}
```

Example: Sorting a set of times

برنامه‌ای یک مجموعه از زمان‌ها را بگیرد و آنها را مرتب کند. هر زمان شامل ساعت، دقیقه و ثانیه است.

```
#include <stdio.h>

struct time{
    int hour;
    int min;
    int sec;
};

/*  1:  t1 > t2, 0:  t1 = t2, -1: t1 < t2  */
int time_cmp(struct time t1, struct time t2){
    if(t1.hour > t2.hour)
        return 1;
    else if(t2.hour > t1.hour)
        return -1;
    else if(t1.min > t2.min)
        return 1;
    else if(t2.min > t1.min)
        return -1;
    else if(t1.sec > t2.sec)
        return 1;
    else if(t2.sec > t1.sec)
        return -1;
    else
        return 0;
}
```

Example: Sorting a set of times

ادامه

```
void time_swap(struct time *t1, struct time *t2){
   struct time tmp;
   tmp = *t1;
   *t1 = *t2;
   *t2 = tmp;
}

/* Find index of max element */
int rec_max(struct time time_arr[], int start, int end){
   int tmp, res;
   if(start == end)
       res = start;
   else{
       tmp = rec_max(time_arr, start + 1, end);
       if(time_cmp(time_arr[start], time_arr[tmp]) >= 0)
           res = start;
       else
           res = tmp;
   }
   return res;
}
```

Example: Sorting a set of times

ادامه

```
/* Recursively sort array from start to end */
void rec_sort(struct time time_arr[], int start, int end){
   int max;
   if(start == end)
       return;

   max = rec_max(time_arr, start, end);
   time_swap(&(time_arr[start]), &(time_arr[max]));
   rec_sort(time_arr, start + 1, end);
}

/* Print Array elements from start to end */
void print_array(struct time time_arr[], int start, int end){

   for(int i = start; i <= end; i++)
       printf("%d:%d:%d, ", time_arr[i].hour, time_arr[i].min,
   time_arr[i].sec);

   printf("\n");
}
```

Example: Sorting a set of times

```
int main(void){
   struct time ta[5] = {{4, 0, 1},
          {6, 1, 0}, {2, 2, 1},
          {6, 4, 7}, {8, 5, 4}};

   print_array(ta, 0, 4);
   rec_sort(ta, 0, 4);
   print_array(ta, 0, 4);

   return 0;
}
```

ادامه

What We Will Learn

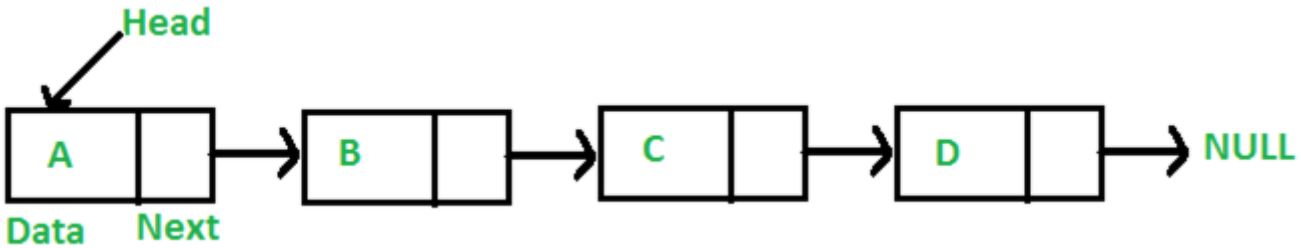
- Introduction
- `struct` definition
- Using `struct`
  - `struct` and Array
  - `struct` and Pointers
  - `struct` and Functions
- Linked-List
- `enum` and `unions`

More Dynamic Data Structures

- In Arrays
  - We know the size of array when you develop code (coding time)
  - We know the size of array when program runs
- What can we do, if we do not know data size even in run time?
  - We use dynamic memory allocation and resize
    - Resizing array has cost and overhead
- What can we do, if we want to add/remove an element to/from middle of the array?
  - We use dynamic memory allocation and resize
    - Resizing array has cost and overhead
  - Is there any other better approach?

Dynamic Data Structures: Linked List

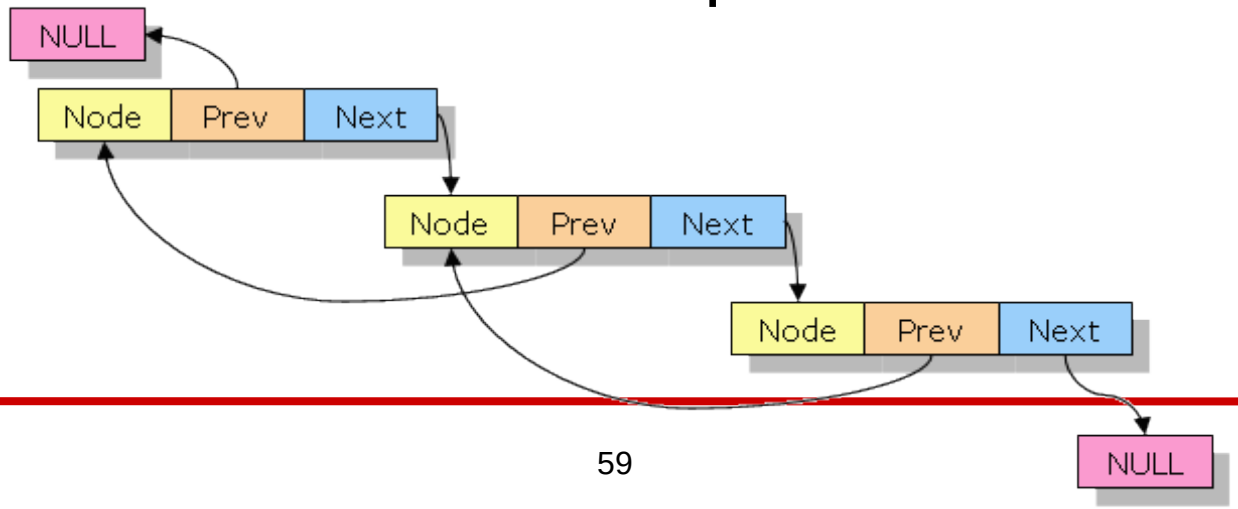
- **Linked list** data structure can be used to implement the dynamic structures
- linked list: Nodes that linked together
  - info (s): Save the information
  - next: Pointer to the next node
  - previous: Pointer to the previous node



58

Dynamic Data Structures: Linked List

- linked list data structure can be used to implement the dynamic structures
- linked list: Nodes that linked together
  - info (s): Save the information
  - next: Pointer to the next node
  - previous: Pointer to the previous node



59

Linked List in C

- linked list is implemented by struct and pointer to struct
- Struct has a member to save the info
- Struct has a pointer to point the next node

```
struct node{
  int info;
  struct node *next;
};
```

Create nodes

- We need a function to create each node in list. The function do
  - 1. Allocate the memory
  - 2. Set the info member
  - 3. Set the next member
  - 4. Return the pointer to new node

Create Node

```
struct node{
   int info;
   struct node *next;
};
struct node * create_node(int i){
   struct node * nn;
   nn = (struct node *) malloc(sizeof(struct node));
   if (nn == NULL)
      return NULL;
   nn->info = i;
   nn->next = NULL;
   return nn;
}
```

Returning pointer!!!
Is it safe?
Why?

Example: 3 Nodes List

```
struct node * list = NULL;

list = create_node(10);

list->next = create_node(20);

list->next->next = create_node(30);
```


Operation on linked list

- Print the list: **`print_list`**
- Add new node to end of list: **`add_end`**
- Add new node to front of list: **`add_front`**
- Insert new node after some node: **`insert_next_node`**
- Delete the first node in list: **`delete_first`**
- Delete the end node in list: **`delete_end`**
- Delete a node from the middle of list: **`delete_next`**

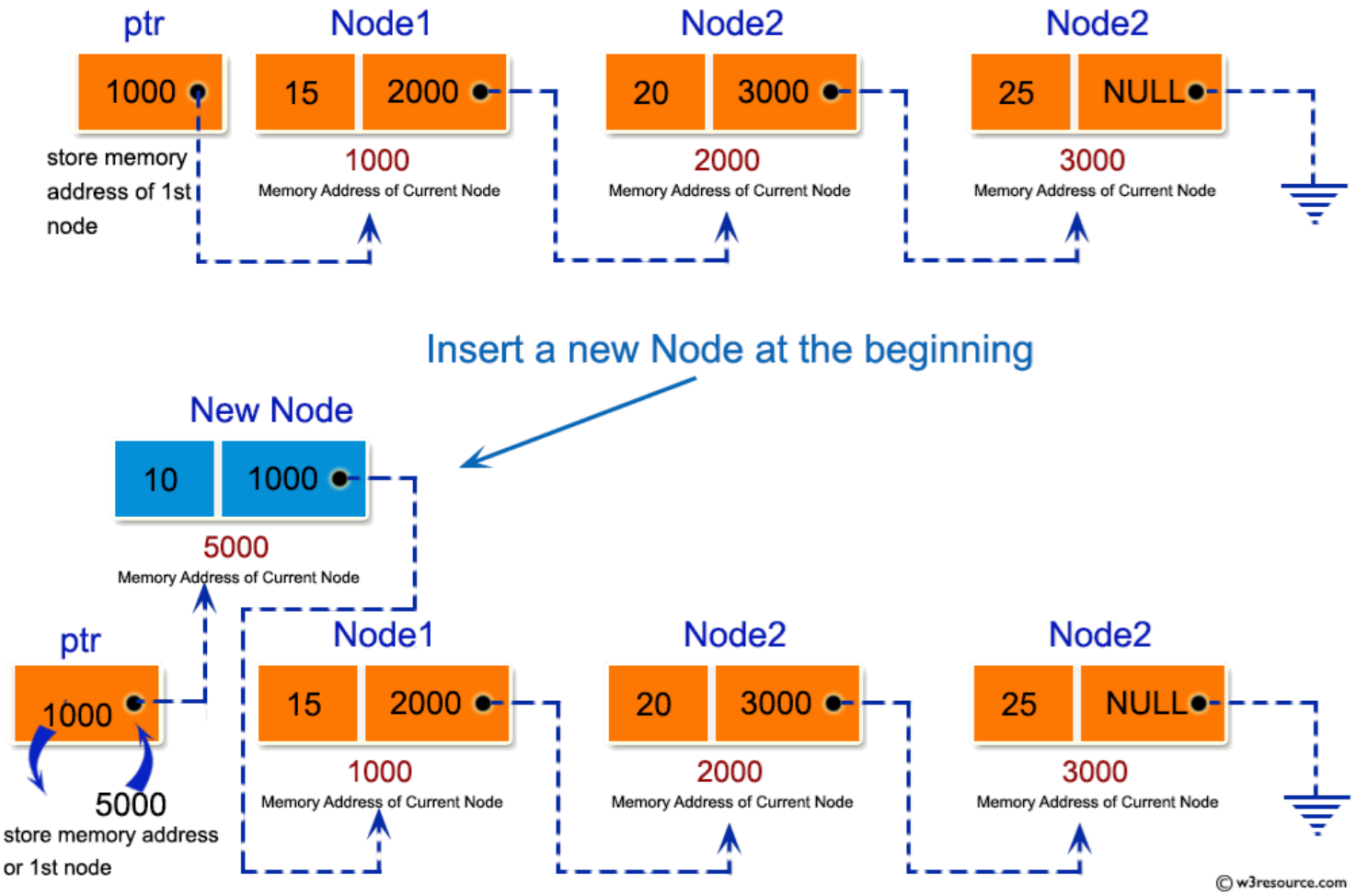
add_end: Add new node to end of list

```
void add_end(struct node *list, struct
  node * new_node){
  struct node *current;
  for(current = list; current-> next != NULL;
  current = current->next);


 current->next = new_node;
  new_node->next = NULL;
}
```

delete_end (if more than 1 nodes)

```
void delete_end(struct node * list){
  struct node * current = list;
  while(current->next->next != NULL)
     current = current->next;

  free(current->next);
  current->next = NULL;
}
```

add_front: Add new node in start of list



`add_front`: Add new node in start of list

```
void add_front_wrong(struct node *list, struct
  node *new_node){

  new_node->next = list;

  list = new_node;

}
```

`add_front`: Add new node in start of list

```
void add_front_wrong(struct node *list, struct
  node *new_node){

  new_node->next = list;

  list = new_node;

}
```

- **Passing Pointers:** The original function should take a pointer to the head pointer of the list (`struct node** list`) instead of a **copy of the head node** (struct node list).
  - This allows us to modify the head pointer of the list to point to the new node.

`add_front`: Add new node in start of list

```
void add_front_wrong(struct node *list, struct
  node *new_node){

  new_node->next = list;

  list = new_node;

}
```

- **Dereferencing Pointers:** When assigning the **new_node** to the list, you need to **dereference** the head pointer with ***list** to update the actual head of the list.

Testing `add_front` function

```
#include <stdio.h>
#include <stdlib.h>
struct node { int info; struct node * next; };

struct node * create_node(int i){
   struct node * nn;
    nn = (struct node *) malloc(sizeof(struct node));
    if (nn == NULL)
       return NULL;
   nn->info = i;
   nn->next = NULL;
   return nn;
};

void print_list(struct node *list){
     struct node * current = list;
     while(current != NULL){
           printf("%p: %d , ", current, current->info);
           current = current->next;
     }
     printf("\n");
}
```

Testing `add_front` function

```
void add_front_wrong(struct node * list, struct node *
   new_node){
     new_node->next = list;
     list = new_node;
}
int main(){
     struct node *tmp, *list = NULL;
     list = create_node(20);
     list -> next = create_node(30);
     tmp = create_node(10);
     print_list(list);
     add_front_wrong(list, tmp);
     print_list(list);
     return 0;
}
          // 0x55d3387d92a0: 20 , 0x55d3387d92c0: 30 ,
          // 0x55d3387d92a0: 20 , 0x55d3387d92c0: 30 ,
```

`add_front`: Add new node in start of list

```
void add_front(struct node **plist, struct
  node *new_node){

  new_node->next = *plist;

  *plist = new_node;

}
main(){
  struct node * list;
  …
  add_front(&list, new_node1);
}
// 0x558486f382a0: 20 , 0x558486f382c0: 30 ,
//0x558486f382e0: 10 , 0x558486f382a0: 20 , 0x558486f382c0: 30 ,
```

Convert array to set using linked list

برنامه‌ای که یک آرایه را بگیرد و با حذف عضوهای تکراری آن، یک لیست پیوند ایجاد کند.

```
#include <stdio.h>
#include <stdlib.h>

struct node{
        int value;
        struct node *next;
};

int in_list(struct node *list, int i){
    struct node *current = list;
    while(current != NULL){
      if(current->value == i)
         return 1;
      current = current->next;
    }
    return 0;
}
```

Convert array to set using linked list

```
void add_front(struct node *new_node, struct node **list){
      new_node->next = *list;
      *list = new_node;
}

void add_end(struct node *new_node, struct node *list){
   struct node *current;
   for(current = list; current-> next != NULL; current =
   current->next);
   current->next = new_node;
   new_node->next = NULL;
}

void print_list(struct node *list){
      struct node * current = list;
      while(current != NULL){
            printf("%d ", current->value);
            current = current->next;
      }
}
```

ادامه

Convert array to set using linked list

ادامه

```
struct node *create_set(int arr[], int size){
        int i;
        struct node *list = NULL;
        for(i = 0; i < size; i++)
              if(in_list(list, arr[i]) == 0){
                    struct node *new_node =
                        (struct node *)malloc(sizeof(struct node));
                    if(new_node == NULL){
                         printf("Cannot create node\n");
                         exit(-1);
                    }
                    new_node->value = arr[i];
                    new_node->next=NULL;
                    if(list == NULL)
                        add_front(new_node, &(list));
                    else
                        add_end(new_node, list);
              }
        return list;
}
```

Convert array to set using linked list

ادامه

```
int main(void){
    int myarr[]={1,2,1,3,1,7,8,2,3,4,11,4,9,9,9,10};
    struct node * mylist =
        create_set(myarr, sizeof(myarr) / sizeof(myarr[0]));

    print_list(mylist);  // 1 2 3 7 8 4 11 9 10

    getchar();
    return 0;
}
```

What We Will Learn

- Introduction
- `struct` definition
- Using `struct`
  - `struct` and Array
  - `struct` and Pointers
  - `struct` and Functions
- Linked-List
- **`enum` and `unions`**

Introduction

- Some data are **naturally ordered**
  - Days of week
  - Months of year
- We want to use the order, *e.g.*
  - The number of visitors per day
  - The salary per month
- We need an array
  - visitors[0] → The number of visitors in Saturday
  - visitors[1] → The number of visitors in Sunday

Introduction

- Enumeration is a mechanism to assign a name for each number
- We can use names instead of numbers
  - More **readable** core
- *E.g.*:

  visitors[saturday], visitors[friday]

  salary[april], salary[june]

enum

- **enum** is used to define a set of names and their corresponding numbers

```
enum tag {name_1, name_2, …, name_N}
```

- **tag** is the enumeration type
  - We use it to define variables
- name_1 = 0
- name_2 = 1
- name_i = (Name_(i-1)) + 1

enum

```
enum week {sat, sun, mon, tue,
  wed, thu, fri};
```

- // sat = 0, sun = 1, mon = 2, …, fri = 6

```
enum year {feb, jan, mar, apr,
  may, jun, jul, aug, sep, oct,
  nov, des};
```

- // feb = 0, jan = 1, …, nov = 10, des = 11

enum

- We can assign the numbers

```
enum week {sat = 1, sun, mon, tue,
  wed, thu, fri};
```

- // sat = 1, sun = 2, mon = 3, …, fri = 7

```
enum condition {False = 0, True,
  No = 0, Yes, Ghalat = 0, Dorost};
```

- // False = No = Ghalat = 0
- // True = Yes = Dorost = 1

enum

- After definition of an enumeration
  - We can use the **tag** to declare variables
  - We can use the names to assign values to the variables

```
enum week {sat, sun, mon, tue,
  wed, thu, fri};

enum week day = sat;

for(day = sat; day <= fri; day++)
```

Example: Read the number of visitors

```
enum week {sat, sun, mon, tue,
  wed, thu, fri};

int visitors[7];

enum week day;

for(day = sat; day <= fri; day++)

    scanf("%d", &visitors[day]);
```

Caution

- C compiler does not check the value is assigned to the **`enum`** variables

```
enum test1 {t1, t2, t3};

enum test2 {t4, t5, t6};

enum test1 t1v = t1;

enum test2 t2v = t4;

if(t1v == t2v) → true

t1v = t5;

t2v = 100;
```

Unions

- The **`union`** keyword in C lets you define a derived data type,
  - Very much similar to the **`struct`** keyword.
- A union data type in C also that allows to store different data types in the **consecutive memory location**.
- Unlike a struct variable, a variable of **union** type, only one of its members can contain a value at any given time.

Union declaration

```
union [union tag] {
    member definition;
    member definition;
    ...
    member definition;
} [one or more union variables];
```

Union memory layout

```
union myunion{
    int a;
    double b;
    char c;
};

sizeof(union myunion)

// 8
```

```
struct mystruct{
    int a;
    double b;
    char c;
};

sizeof(struct mystruct)

// 24
```

Same memory location, can be used to store multiple types of data

Union example

```
#include <stdio.h>
#include <string.h>
union Data {
   int i;
   float f;
   char str[20];
};
int main() {
   union Data data;
   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");
   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);
   return 0;
}
```

```
// 1917853763
// 41223605803277948604527599943 68.000000
// C Programming
```

Union example

```
#include <stdio.h>
#include <string.h>
union Data {
   int i;
   float f;
   char str[20];
};
int main(){
   union Data data;
   data.i = 10;
   printf( "data.i : %d\n", data.i); // 10
   data.f = 220.5;
   printf( "data.f : %f\n", data.f); // 220.500000
   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str); // C Programming
   return 0;
```

Common Bugs

- The last “NULL” in Liked-list is very important
  - Always keep it
- Operation of linked-list has many exceptions
  - When list is empty
  - When we want to add to the first of list
  - …

Reference

- Reading Assignment: Chapter 10 and Sections 12.1-12.4 of “C How to Program”